Top-Down Parsing

CS164
Lecture 6-7

Review

- A parser consumes a sequence of tokens *s* and produces a parse tree
- Issues:
  - How do we recognize that $s \in L(G)$?
  - A parse tree of *s* describes how $s \in L(G)$
  - Ambiguity: more than one parse tree (interpretation) for some string *s*
  - Error: no parse tree for some string *s*
  - How do we construct the parse tree?

Ambiguity

- Grammar

  E → E + E | E * E | (E) | int

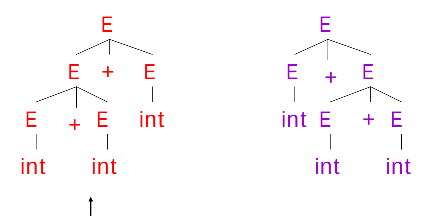
- Strings

  int + int + int

  int * int + int

Ambiguity. Example

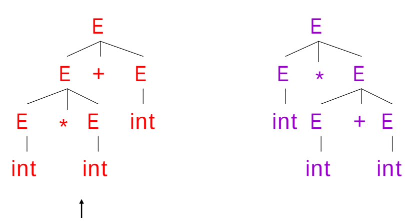
The string int + int + int has two parse trees



+ is left-associative

Ambiguity. Example

The string int * int + int has two parse trees



* has higher precedence than +

Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is bad
  - Leaves meaning of some programs ill-defined
- Ambiguity is common in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously

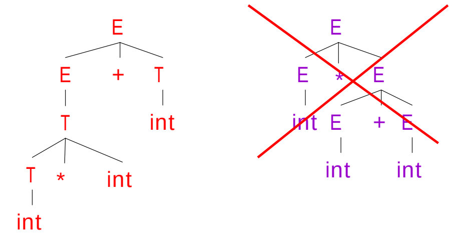
  E → E + T | T

  T → T * int | int | (E)

- Enforces precedence of * over +
- Enforces left-associativity of + and *

Ambiguity. Example

The int * int + int has ony one parse tree now



Ambiguity: The Dangling Else

- Consider the grammar

  E → if E then E

  | if E then E else E

  | OTHER

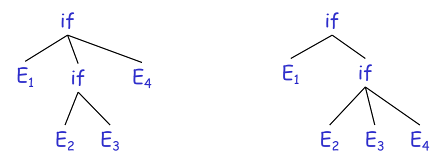
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

- else matches the closest unmatched then
- We can describe this in the grammar (distinguish between matched and unmatched "then")

  E → MIF /* all then are matched */

  | UIF /* some then are unmatched */

  MIF → if E then MIF else MIF

  | OTHER

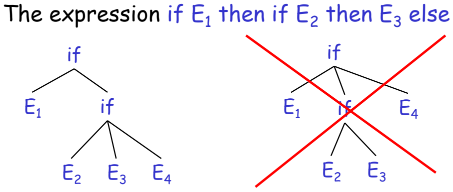
  UIF → if E then E

  | if E then MIF else UIF

- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then E_3 else E_4



- A valid parse tree (for a UIF)
- Not valid because the then expression is not a MIF

Ambiguity

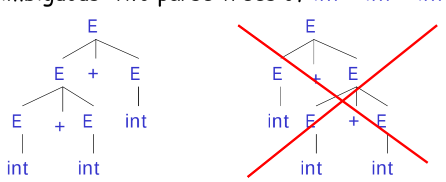
- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

Precedence and Associativity Declarations

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow <u>precedence and associativity declarations</u> to disambiguate grammars
- Examples …

Associativity Declarations

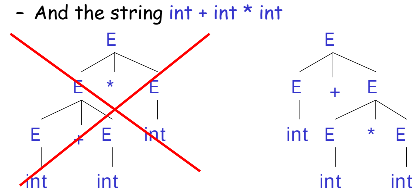
- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int



- Left-associativity declaration: %left +

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int



- Precedence declarations: %left +
  %left *

Review

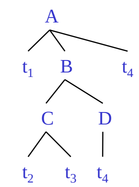
- We can specify language syntax using CFG
- A parser will answer whether $s \in L(G)$
- … and will build a parse tree
- … and pass on to the rest of the compiler
- Next:
  - How do we answer $s \in L(G)$ and build a parse tree?

Top-Down Parsing

Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:

  t_1 t_2 t_3 t_4 t_5



- The parse tree is constructed
  - From the top
  - From left to right

Recursive Descent Parsing

- Consider the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow (E) \mid int \mid int * T$
- Token stream is: int * int
- Start with top-level non-terminal E

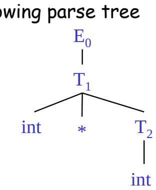
- Try the rules for E in order

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1 + E_2$
- Then try a rule for $T_1 \rightarrow (E_3)$
  - But (does not match input token int
- Try $T_1 \rightarrow$ int . Token matches.
  - But + after T_1 does not match input token *
- Try $T_1 \rightarrow$ int * T_2
  - This will match but + after T_1 will be unmatched
- Have exhausted the choices for T_1
  - Backtrack to choice for E_0

Recursive Descent Parsing. Example (Cont.)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for T_1
  - And succeed with $T_1 \rightarrow$ int * T_2 and $T_2 \rightarrow$ int
  - With the following parse tree



Recursive-Descent Parsing

- Parsing: given a string of tokens t_1 t_2 ... t_n, find its parse tree
- Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: t_1 t_2 … t_k A …
  - Try all the productions for A: if $A \rightarrow BC$ is a production, the new fringe is t_1 t_2 … t_k B C …
  - Backtrack when the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S\ a$:
  - In the process of parsing S we try the above rule
  - What goes wrong?

- A left-recursive grammar has a non-terminal S

  $S \rightarrow^{+} S\alpha$ for some α

- Recursive descent does not work in such cases
  - It goes into an infinite loop

Elimination of Left Recursion

- Consider the left-recursive grammar
  $S \rightarrow S\ \alpha \mid \beta$
- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion
  $S \rightarrow \beta\ S'$
  $S' \rightarrow \alpha\ S' \mid \varepsilon$

Elimination of Left-Recursion. Example

- Consider the grammar
  $S \rightarrow 1 \mid S\ 0$ (β = 1 and α = 0)

  can be rewritten as
  $S \rightarrow 1\ S'$
  $S' \rightarrow 0\ S' \mid \varepsilon$

More Elimination of Left-Recursion

- In general
  $S \rightarrow S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as
  $S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$
  $S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

General Left Recursion

- The grammar
  $S \rightarrow A\ \alpha \mid \delta$
  $A \rightarrow S\ \beta$
  is also left-recursive because
  $S \rightarrow^{+} S\ \beta\ \alpha$
- This left-recursion can also be eliminated
- See book, Section 4.3 for general algorithm

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- Often, we can avoid backtracking …

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
- In practice, LL(1) is used

LL(1) Languages

- In recursive-descent, for each non-terminal and input token there may be a choice of production
- LL(1) means that for each non-terminal and token there is only one production that could lead to success
- Can be specified as a 2D table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

Predictive Parsing and Left Factoring

- Recall the grammar
  E → T + E | T
  T → int | int * T | (E)
- Impossible to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- A grammar must be left-factored before use for predictive parsing

Left-Factoring Example

- Recall the grammar
  E → T + E | T
  T → int | int * T | (E)
- Factor out common prefixes of productions
  E → T X
  X → + E | ε
  T → (E) | int Y
  Y → * T | ε

LL(1) Parsing Table Example

- Left-factored grammar
  E → T X   X → + E | ε
  T → (E) | int Y   Y → * T | ε
- The LL(1) parsing table ($ is a special end marker):

	int	*	+	(	)	$
T	int Y			(E)		
E	T X			T X		
X			+ E		ε	ε
Y		* T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production E → T X
  - This production can generate an int in the first place
- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - We'll see later why this is so

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  - Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And choose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S,$> and next (pointer to tokens)
repeat
  case stack of
    <X, rest>  : if T[X,*next] = Y1…Yn
                   then stack ← <Y1… Yn rest>;
                   else  error ();
    <t, rest>  : if t == *next ++
                   then  stack ← <rest>;
                   else error ();
until stack == < >
```


LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing Parsing Tables

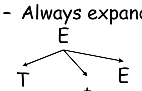
- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

- We want to generate parsing tables from CFG

Top-Down Parsing. Review

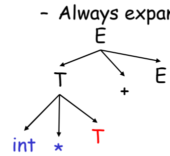
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

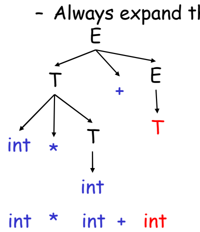


- The leaves at any point form a string βAγ
  - β contains only terminals
  - The input string is βbδ
  - The prefix β matches
  - The next token is b

int * int + int

Top-Down Parsing. Review

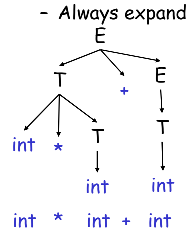
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Constructing Predictive Parsing Tables

- Consider the state $S \to^* \beta A \gamma$
  - With b the next token
  - Trying to match $\beta b \delta$

There are two possibilities:

1. b belongs to an expansion of A
   - Any $A \to \alpha$ can be used if b can start a string derived from α

     In this case we say that $b \in First(\alpha)$

Or…

Constructing Predictive Parsing Tables (Cont.)

2. b does not belong to an expansion of A
   - The expansion of A is empty and b belongs to an expansion of γ (e.g., $b\omega$)
   - Means that b can appear after A in a derivation of the form $S \to^* \beta A b \omega$
   - We say that $b \in Follow(A)$ in this case

   - What productions can we use in this case?
     - Any $A \to \alpha$ can be used if α can expand to ε
     - We say that $\varepsilon \in First(A)$ in this case

Computing First Sets

Definition $First(X) = \{ b \mid X \to^* b\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$

1. $First(b) = \{ b \}$

2. For all productions $X \to A_1 \ldots A_n$
   - Add $First(A_1) - \{\varepsilon\}$ to $First(X)$. Stop if $\varepsilon \notin First(A_1)$
   - Add $First(A_2) - \{\varepsilon\}$ to $First(X)$. Stop if $\varepsilon \notin First(A_2)$
   - …
   - Add $First(A_n) - \{\varepsilon\}$ to $First(X)$. Stop if $\varepsilon \notin First(A_n)$
   - Add ε to $First(X)$

First Sets. Example

- Recall the grammar

  $E \to T X$ $\quad X \to + E \mid \varepsilon$

  $T \to (E) \mid int\ Y$ $\quad Y \to * T \mid \varepsilon$

- First sets

First(( ) = { (}	First(T) = {int, (}
First( )) = {) }	First(E) = {int, (}
First(int) = { int }	First(X) = {+, ε }
First(+) = { + }	First(Y) = {*, ε }
First(*) = { * }	

Computing Follow Sets

Definition $Follow(X) = \{ b \mid S \to^* \beta X b \omega \}$

1. Compute the First sets for all non-terminals first
2. Add \$ to Follow(S) (if S is the start non-terminal)

3. For all productions $Y \to \ldots X A_1 \ldots A_n$
   - Add $First(A_1) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_1)$
   - Add $First(A_2) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_2)$
   - …
   - Add $First(A_n) - \{\varepsilon\}$ to Follow(X). Stop if $\varepsilon \notin First(A_n)$
   - Add Follow(Y) to Follow(X)

Follow Sets. Example

- Recall the grammar

$E \to T X \qquad X \to + E \mid \varepsilon$

$T \to (E) \mid int\ Y \qquad Y \to * T \mid \varepsilon$

- Follow sets

Follow(E) = {), \$}

Follow(X) = {\$,) }

Follow(Y) = {+,) , \$}

Follow(T) = {+,) , \$}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \to \alpha$ in G do:
  - For each terminal $b \in First(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\alpha \to^* \varepsilon$, for each $b \in Follow(A)$ do
    - $T[A, b] = \alpha$

Constructing LL(1) Tables. Example

- Recall the grammar

$E \to T X \qquad X \to + E \mid \varepsilon$

$T \to (E) \mid int\ Y \qquad Y \to * T \mid \varepsilon$

- Where in the line of Y we put $Y \to * T$?
  - In the lines of First(*T) = { * }

- Where in the line of Y we put $Y \to \varepsilon$?
  - In the lines of Follow(Y) = { \$, +,) }

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - <u>And in other cases as well</u>
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables

Review

- For some grammars there is a simple parsing strategy
  - Predictive parsing (LL(1))
  - Once you build the LL(1) table, you can write the parser by hand

- Next: a more powerful parsing strategy for grammars that are not LL(1)